Chapter 11: Bitwise Instructions

11.1 Shift Instructions

The next class of instructions pushes bits around as if they were billiard balls. The billiard ball analogy breaks down, of course, because bits may "disappear" from the end of a register in some cases, or, in other cases, be created. There are three types of shift instructions:

- The so-called "logical" shift instructions move bits in strict order from left to right or right to left. Bits which are shifted off the end of a register are lost forever (going into the proverbial "bit bucket"). Bits added at the other end are always zero (clear).
- Circular shifts treat the register as if it were connected at each end. Bits removed on one end reappear at the other end, maintaining their left-to-right order.
- An algebraic shift moves the register contents to the right. The starting value of the leftmost (that is, the sign) bit is repeated by the shift count as the rest of the data is moved to the right. This operation is called an algebraic shift because the sign bit propagation just described extends the sign. An example later will make this point clearer. Data on the right of the register is lost.

For each of these types, there are shifts which operate on both single registers and pairs of registers. Logical and circular shifts can be found in two varieties, left and right. An algebraic shift shifts to the right only.

The CPU uses only 7 bits of U, for a maximum value of 127. If a shift count over 72 is specified, the results of the operation are undefined.

11.2 Single Shifts

11.2.1 Logical Shifts

Logical shifts, left and right, are quick ways to multiply or divide by powers of two. They are also quick, but not the only, ways of isolating bit-width (non-partial-word) fields. SSL (Single Shift Logical) shifts the contents of the named a register right by U bits. Since it is a logical shift, the bits shifted to the right of the register are lost. Here is an example of SSL:

```
L,U         A2,256      .
SSL         A2,2        .
L,U         A5,27       .
SSL         A5,1        .
```

Can you figure out the results in A2 and A5? Two to the second power is 4, and 256 divided by 4 is 64, the result in A2. The second example is trickier. The short answer is that an integer division of 27 by 2 discards the remainder 1. The result in A5 (the quotient) is 13.

The bitwise answer takes a bit of figuring (pun intended). First, 27 decimal converts to 033 octal. Writing down the binary equivalent of 033, we get $011{,}011_2$. A right shift of 1 bit discards the

rightmost binary digit. Regrouping the binary digits obtains $001,101_2$, or 015 octal, which is 13 decimal.

If the shift count is not known at coding time, you can put the shift count (or part of it) in an index register. The final shift count is U, the total of the u field and XM(X_x). Here is how you can code a variable shift count:

```
          L           X5,SCOUNT   .
          SSL         A1,2,X5     .
```

In this example, the shift count will be 2 plus the contents of SCOUNT.

An LSSL (Left Single Shift Logical) shifts the contents of the named A register left by U bits. It is a handy way to multiply by a power of 2. To sharpen your talents with octal, verify the result of this example:

```
SCOUNT    +           1           .
          L           A3,SCOUNT   .
          L,U         A2,0273461      .
          LSSL        A2,1,A3     .
```

Did you get 01356304? Here's how you should have gotten it:

```
Original A2:          0  0  0  0  0  0  2  7  3  4  6  1
In binary:       000000000000000000010111011100110001
Add 2 zeroes on right:
                 000000000000000000010111011100110001 00
(Shift count is 2)
Regroup:         000000000000000000010111011100110001 00
In octal:             0  0  0  0  0  1  3  5  6  3  0  4
```

An SSA (Single Shift Algebraic) allows you to divide signed numbers by powers of 2, preserving the sign. In a more exotic application, you could use it to create signed fields of arbitrary widths such as 7, 13, or 26 bits — widths not supported by signed hardware partial words. This is a more everyday example:

```
          L,XU        A0,-100     .
          SSA         A0,2        .
```

The result in A0 is -25. Here it is in octal:

```
(Positive 100):       0  0  0  0  0  0  0  0  0  1  4  4
Original A0:          7  7  7  7  7  7  7  7  7  6  3  3
In binary:       111111111111111111111111110011011
After shift:     111111111111111111111111111100110
Ending A0:            7  7  7  7  7  7  7  7  7  7  4  6
Ones complement:      0  0  0  0  0  0  0  0  0  0  3  1
```

The last step (for our convenience, not done by the instructions) is to get the unsigned value of the result, 25.

11.2.2 Circular Shifts

Circular shifts are a bit more fun. An SSC (Single Shift Circular) shifts A_a right by U bits. The bits shifted off to the right are added on the left in their original order. Here is a simple example:

```
L           A1,(+ 0123456654321)  .
SSC         A1,3                  .
```

The result in A1 is 0112345665432. Note that a shift count of a multiple of 3 preserves the octal digits. (Write the example in binary if you need convincing.)

A shift count of 18 reverses the left and right halves of the register (XI and XM for A0 through A3). You can play quite a shell game with registers, possibly confusing yourself in the process! (This is where you must have clear inline comments.) Try this variation of the last example:

```
L           A1,(+ 0123456654321)  .
SSC         A1,18.
```

The result in A1 is 0654321123456.

LSSC (Left Single Shift Circular) is the same story, only it shifts to the left. This example is left as an exercise for you (hint: the leftmost and rightmost digits are both 2):

```
L           A1,(+ 0123456654321)  .
LSSC        A1,1                  .
```

11.3 Logical Instructions

11.3.1 AND - Logical Multiplication

The instructions AND, OR, XOR, and MLU are called "logical" because they operate on the individual bits of the operands, rather than on the arithmetic value. For example, for each bit position of A_a, the results of an AND instruction are defined in this "truth table":

		Operand 2	
	AND	0	1
0	0	0	0
p			
1	1	0	1

In other words, the result is only a 1 bit if both operands are 1.

The AND instruction logically ANDs the bits in Aa and U, and puts the result in Aa+1. While putting the result in a different register may seem strange at first, it comes in handy testing the same mask for multiple bits. The original value is preserved. Here is an example:

```
            L           A5,STATUS   .
            AND,U       A5,COND1    .
            JNZ         A6,DOCON1   .
            AND,U       A5,COND2    .
            JNZ         A6,DOCON2   .
```

And we could repeat the AND/jump pair as many times as necessary.

11.3.2 OR - Logical Addition

The OR instruction uses this truth table:

		Operand 2	
	OR	0	1
Operand 1	0	0	1
	1	1	1

In other words, the result is a 1 bit if either operand is a 1 bit.

The OR instruction logically ORs the contents of A_a and U and places the result in A_{a+1}. Like AND, it preserves A_a, allowing repeated operations on the same value. While AND essentially disassembles a bit mask, OR is most useful building up a mask. This code sample shows one way to do it:

```
            SZ          A5          . INITIAL VALUE OF MASK
            TZ          FLAG1       . FLAG1 ZERO?
            J           DNC050      .  NO,DNC050  YES,
            OR,U        A5,COND1    . ADD COND1 BITS
            L           A5,A6       . PUT MASK BACK
DNC050      .
            TNZ         FLAG2       . FLAG2 ZERO?
            J           DNC060      .  YES,DNC060  NO,
            OR,U        A5,COND2    . ADD COND2 BITS
            L           A5,A6       . PUT MASK BACK
DNC060      .
```

Each OR,U adds bits to the mask. Can you see why the code must copy A6 back to A5 after each OR instruction? The ORs replace the contents of A6, thus losing the result of the previous OR operation.

11.3.3 XOR - Logical Exclusive OR (Logical Difference)

Last in the trio of basic logic operations is the XOR instruction. The truth table for XOR is:

	XOR	Operand 2: 0	1
0	0	0	1
P			
1	1	1	0

In other words, the result is a 1 bit if only one of the operands is a 1, otherwise the result is a 0. This instruction can be used to flip all the bits in a mask, as in this example:

```
MASK1       +           01274       .
            L           A7,MASK1    . GET OLD MASK
            XOR         A7,(- 0)    . FLIP ALL BITS
            S           A8,MASK1    . REPLACE MASK
```

What is the operand of the XOR instruction? The ones complement of zero (that is, minus zero) is a word of all 1 bits, 0777777777777. XORing MASK1 with a mask of all 1 bits switches each 0 bit to a 1 bit, and each 1 bit to a zero.

Granted, there is a simpler way to get the complement of MASK1. A simple LNA A7,MASK1 would flip each bit in the same manner. Suppose you needed to flip only bits 2 (33) and 27 (8)? This is a less trivial use of XOR:

```
MASK1       +           01274       .
            L           A7,MASK1    . GET OLD MASK
            XOR         A7,(+ 1*/(35-2)+1*/(35-27))  .
                                    . FLIP BITS 2 AND 27
            S           A8,MASK1    . REPLACE MASK
```

You can choose to put a constant operand for AND, OR, or XOR (as for any instruction), in one of two places:

- If it will fit in 18 bits, make it an immediate operand, as in:

```
            AND,U       A4,045      .
```

- If it will not fit in 18 bits, you must put the operand in a separate word, usually a literal, as in:

```
            XOR         A7,(+ 1*/(35-2)+1*/(35-27))  .
                                    . FLIP BITS 2 AND 27
```

If you have specified a 1 shifted by more than 17, for example, the operand will not fit in 18 bits.

(Note: The */ operator will be explained in more detail in Chapter 17. It shifts the amount on the left, like the 1s above, to the left by a number of bit positions given by the right amount, like the 35-2 above. You can think of it multiplying by a power of 2. While we can figure out by hand that bit 2 (33) is 0100000000000, it is much easier to code */ expression and let MASM do what it does best.)

11.4 Bitwise Test Instructions

Test Odd Parity (TOP) and Test Even Parity (TEP) logically AND the contents of A_a and the contents of U. They either skip NI or execute it, depending on how many bits are set in the result.

- Test Odd Parity skips NI if the result has odd parity, that is, if there is an odd number of bits set.
- Test Even Parity skips NI if the result has even parity, that is, an even number of bits are set.

In practice, about all they are good for is testing one bit on (odd parity) or off (even parity). Here is a sequence to test MASK1 above for bit 2 (33) set:

```
L           A1,MASK1    .
TOP         A1,(+ 1*/33)  . BIT 2 SET?
J           RNM060      .  NO,RNM060  YES,
```

Note that since the test value used by the TOP is wider than 18 bits, it can not use an immediate operand (partial-word modifier of U). If you need to reverse the test and jump if bit 2 (33) is set rather than clear, use TEP:

```
L           A1,MASK1    .
TEP         A1,(+ 1*/33)  . BIT 2 CLEAR?
J           RNM060      .  NO,RNM060  YES,
```

11.5 MLU - Masked Load Upper

The formal description of this instruction in an instruction chart is a little intimidating:

[(U) AND (R2)] OR [(A_a) AND NOT (R2)] -> A_{a+1}

But the instruction is not hard to use. One hint to this operation is the fact that R2 is the mask register. Further, (U) means the contents of U.

Masked Load Upper replaces a portion of A_a with the contents of U and places the result in A_{a+1}. The portion of A_a which is replaced is controlled by the contents of R2. Before we get to example code, it is important to dissect the formal description so you can refer to this instruction when you need it.

- The construct [(U) AND (R2)] means that for each bit set in R2, a bit of (U) will be in the result.
- And the construct [(A_a) AND NOT (R2)] means that for each bit *clear* (not set) in R2, a bit of A_a will be present in the result.
- Finally, the OR in the middle of the description means that the two intermediate results are combined to make the final result in A_{a+1}.

If you are still confused, here is an example. To make it a little easier, the field in question will be S5. (Normally, you will want to use the j field to access a partial word such as S5.)

```
MFIELD      +           01,5,27,32,0,056  . STARTING VALUE
            L,U         R2,07700    . SET S5 MASK.
            L           A2,MFIELD   . GET FIRST OPERAND.
            MLU,U       A2,05       . INSERT 5 IN S5 OF MFIELD
            S           A3,MFIELD   . REPLACE MFIELD
```

Note that the contents of A2 are preserved. The word "upper" in the instruction name should be a hint to you that the result goes to the "upper" register, that is, A_{a+1}.

You will need this instruction most often for bit-width fields such as defined by a FORM directive. But while the example uses a mask of 6 contiguous bits, the instruction has no such requirement. You could use MLU to take the contents of bits 1, 5, and 24 of one word and massage them into another word. In fact, the operand words don't even need to line up, bit for bit. You can use ordinary partial words (S1, H2, and so forth) on the MLU instruction. In that case, the (U) operand is simply the specified portion of the contents of U.

11.6 Double Word Shifts

The double-word shift instructions treat a pair of A registers as a single 72-bit register. Code to work on 1- and 2-word Fieldata character strings uses these instructions. (Character handling is covered in more detail in Chapter 21.) These are the instructions:

- DSL (Double Shift Logical) - shift A_a and A_{a+1} right by U bit positions and zero fill on the left.
- LDSL (Left Double Shift Logical) - shift A_a and A_{a+1} left by U bit positions and zero fill on the right.
- DSC (Double Shift Circular) - shift A_a and A_{a+1} circularly right by U bit positions.
- LDSC (Left Double Shift Circular) - shift A_a and A_{a+1} circularly left by U bit positions.
- DSA (Double Shift Algebraic) - the contents of A_a and A_{a+1} are shifted right by U bit positions, extending the sign bit as described for Single Shift Algebraic.

DSL, with a 36-bit shift count, is excellent for setting up a DI instruction, especially when dividing the same quantity by successive divisors.

11.7 LSC - Load Shift and Count

There are two more bitwise instructions, Load Shift and Count (LSC) and Double Load Shift and Count (DLSC). LSC loads A_a from U and shifts the contents of A_a circularly left until bit 0 (35) is not equal to bit 1 (34). The number of bit positions by which Aa was shifted is transferred to A_{a+1}. (Recall that bits 0 (35) and 1 (34) are the leftmost two bits in a register.) DLSC does the same operation on 2 words at a time; the shift count goes into A_{a+2}.

While at least one Storage and Processor reference manual lists LSC and DLSC as normalizing instructions (that is, used for adjusting fractions to a standard format), you can also use them to count bits in a multi-word bit table. The master bit tables which record allocation on each disk unit are

examples of places you can use LSC. A zero bit is an unallocated track; a one bit is an allocated or unusable track. Such applications are beyond the scope of this text. Just keep LSC and DLSC in mind when you approach large bit tables.

11.8 Strategies For Accessing Bit-Width Fields

If you need to massage a value into a field of arbitrary width, there are a variety of ways, depending on the exact circumstances. Your choice should take into account the registers which are available, where the operands are and which fields are going to end up where.

For example, the parameter for ER PRINT$ uses three fields, respectively 12, 6 and 18 bits. (For the moment, assume that they are not on partial-word boundaries.) These fields are the line count, the buffer size in words and the buffer address. This code assumes that the length of the buffer has just been calculated in A0:

```
LBUFF     RES       22          . PRINT LINE BUFFER
          A,U       A0,0100     . ADD LINE COUNT OF 1
          LSSL      A0,18       . MAKE ROOM FOR BUFF ADDRESS
          A,U       A0,LBUFF    . INSERT BUFFER ADDRESS
          ER        PRINT$      . PRINT THE LINE
```

This simple example needs only one shift. Use these guidelines coding instructions which access bit-width fields:

- For speed of execution, try to use the fewest possible instructions to assemble or break apart a field, within reason. On the other hand, straining 4 hours to save 120 nanoseconds per execution is false economy.
- Your choice of instructions should be easy to understand, even if it is not absolutely the fastest.
- A shift count which is directly dependent on the width of a field should show an operand of 6+2, 36-6 or whatever, as needed to make the operation clear.

Here is an example of the latter. It isolates the function (op) code field of an instruction word:

```
IWORD     L,H2      A5,CELL15,X2 . TEST INSTRUCTION WORD
OPCODE    +         0            . OP CODE GOES HERE
          L         A2,IWORD     . GET INSTRUCTION WORD
          SSL       A2,36-6      . ISOLATE OP CODE
          S         A2,OPCODE    . SAVE OP CODE ALONE
```

The shift count 36-6 helps communicate the fact that the width of the field is 6, and MASM understands that the shift count is 30. Directly coding 30 for the shift count may cause mental gymnastics in the reader to figure out what is going on. (Note that this example code destroys the remainder of the instruction word in A2.)

For a less trivial example, assume that your code must break apart all the fields of an instruction word and store them separately. (This operation might be necessary for a formatted dump

of instruction words.) Doing this efficiently requires a bit of concentration to keep track of what is in each register. Here is one way to do it:

```
IWORD     L,H2       A5,CELL15,X2  . TEST INSTRUCTION WORD
OPCODE    +          0             . OP CODE (F) GOES HERE
JDESIG    +          0             . J FIELD GOES HERE
AREG      +          0             . A FIELD GOES HERE
XREG      +          0             . X FIELD GOES HERE
HIBITS    +          0             . H AND I BITS GO HERE
UFIELD    +          0             . U FIELD GOES HERE
.
          L          A4,IWORD      . GET INSTRUCTION WORD
          DSL        A4,16         . PUT U FIELD IN A5
          SSL        A5,36-16      . RIGHT JUSTIFY U
          S          A5,UFIELD     . SAVE U FIELD
          DSL        A4,2          . PUT HI BITS IN A5
          SSL        A5,36-2       . RIGHT JUSTIFY HI
          S          A5,HIBITS     . SAVE HI BITS
          DSL        A4,4          . PUT X FIELD IN A5
          SSL        A5,36-4       . RIGHT JUSTIFY X
          S          A5,XFIELD     . SAVE X FIELD
          DSL        A4,4          . PUT A FIELD IN A5
          SSL        A5,36-4       . RIGHT JUSTIFY A
          S          A5,AREG       . SAVE A FIELD
          DSL        A4,4          . PUT J FIELD IN A5
          SSL        A5,36-4       . RIGHT JUSTIFY J
          S          A5,JDESIG     . SAVE J FIELD
          DSL        A4,6          . PUT OP CODE IN A5
          SSL        A5,36-6       . RIGHT JUSTIFY OP CODE
          S          A5,OPCODE     . SAVE OP CODE
```

One way to double-check this kind of code is to add up the shift counts. They should total the same number as the reference manual's total—in this case, 36.

Exercises

E11-1. For a moment, play as if you're the Exec. Assume A0 has the parameters for an ER PRINT$. Separate those parameters and store them into the whole words LINES, WORDS and BUFFER. Storing A0 into memory and using partial-word modifiers to load them back is not allowed.

E11-2. Divide the whole word SECTORS by 64 using only a shift. (Hint: 64 is a power of 2.)

E11-4. Code a sequence to jump to P9UNIT if bit 9 (26) is set in A4.

E11-5. The cells GOOD10 and GOOD20 each have full-word masks. OR them together and store into GOODALL.

E11-6. Chapter 22 will discuss more ERs, including DATE$. DATE$ puts the date and time into A0 and A1 in Fieldata. The date is in A0 as if you had coded

```
            $FDATA
NOWDAT      +           '07','04','90'  .
NOWPRT      RES         2               . OUTPUT BUFFER FOR EXERCISE
            L           A0,NOWDAT       .
```

for July 4, 1990. Using shifts, and, possibly, MLU, code a sequence to move NOWDAT to NOWPRT with slashes inserted in the date, that is, 07/04/90. (It is dirty pool for this exercise to code separate partial-word instructions for each character.)

E11-7. Given the words with separate instruction word fields in the last example of this chapter, code a sequence to reassemble an instruction word.

E11-8. Given this memory location:

```
PARTS5      +           005010101312  .
```

code a sequence to obtain Q3 of the word without using a partial word modifier.

E11-9. For "extra credit" (this may take a while), take this line:

```
MASTER      +           0502762431005  .
```

and code a sequence to count the number of bits set in it. The easy way counts one bit at a time. The "less easy" way is to use LSC. Please try it both ways. Do not let your code take advantage of any particular bits set or clear in MASTER.

Chapter 12: Basic Executive Requests

12.1 Introduction

Writing assembly language without operating system calls is a little like using a calculator in a dark room: you may push the right buttons, but no one will know the answer. Executive Request is actually only a single instruction. It creates an interrupt, which sort of taps OS 1100 on the shoulder. The operating system looks for the source of the interrupt (your job), and looks at the u field of the instruction to determine which operation you have requested.

At assembly time, you supply the name of the Executive Request (PRINT$, READ$, or other). MASM does not recognize any particular ER. (You will find that the u field of an ER instruction is zero in an assembly listing.) MASM simply places an external reference for the symbol READ$ or whatever into the output relocatable. A system library element defines the ER names at collection time. (The collector inserts the ER index in the u field of the instruction.) The collector automatically knows to look in the system libraries for the ER definitions; you don't have to take any special action. (Don't worry about U flags on ER instructions.)

Some Executive Requests transfer images to and from symbiont files like READ$ and PRINT$. Other symbiont ERs transfer images to and from user files as if they were READ$ or PRINT$. There is a whole class of ERs which start, end, and synchronize activities. (This book will deal with only a few of these). Input/output ERs ask OS 1100 to initiate data transfers between your program and a peripheral. I/O ERs are beyond the scope of this book. Others get information about the system configuration, the contents of the system file directory and so forth.

This chapter shows how to get data in and out of your program in the simplest fashion—via the symbionts—and how to end execution. Chapter 22 will tell you more about the interface between your program and OS 1100.

Executive Requests fall into three classes, according to the way parameters are passed. Examples are shown in parentheses:

- No parameters required. (Exit ERs)
- The only parameter required is passed in a single register. (READ$, PRINT$)
- A register (usually A0) contains the address of a packet, which contains a buffer address, a line count or other things. (READA$, PRINTA$)

12.2 Exit ERs

After your program has done its work, it needs to tell the operating system it is finished, as a Fortran STOP statement does. There are four ERs to do that. The difference between them is how OS 1100 cleans up after the program. The system normally saves a copy of your program to a file called DIAG$. The PMD (Post Mortem Dump) processor reads DIAG$ to give you a dump, if there is information to produce the dump.

Here are the exit ERs, in increasing order of severity:

- EXIT$ - this does a normal exit. The program "image" is written to DIAG$ as normal, allowing a PMD.

- ERR$ - this ER causes an error exit of the requesting activity. The program image is written to DIAG$.

- EABT$ - this ER error-terminates the requesting activity and all other activities of the program. The program image is written to DIAG$.

- ABORT$ - this ER error-terminates the requesting activity and all other activities of the program. It also prevents the write to DIAG$, making a PMD impossible.

(If you are not sure what an "activity" is, don't worry. Your program has only one until you do something to create others, and those techniques are beyond the scope of this text.)

No parameters are required with any of these ERs. Here is how to code an EXIT$ request:

```
ER          EXIT$ .
```

Code calls on the other exit ERs similarly; just substitute the other names for EXIT$ in the instruction as shown. Note that an ER EXIT$ is just another instruction; it does *not* signal the end of source images to MASM.

Because of the action the Exec takes on each of these ERs, you need to choose the ER carefully. Here are suggested uses for each:

- EXIT$ - use this for any normal termination.

- ERR$ - your program has detected an error condition and cannot continue. But you still want to be able to look at the PMD.

- EABT$ - you will use this ER mostly in a multi-activity program when one activity encounters a fatal error condition and you want every other activity terminated, too. In a single activity program (like normal Cobol or Fortran), EABT$ takes the same action as ERR$. You still want a PMD to be possible. (Multiple activities are beyond the scope of this book.)

- ABORT$ - your program has detected a fatal error condition, such as an illegal parameter supplied by a user. You want the program to quit immediately and a PMD is unnecessary. All other activities in the program are error terminated, too.

12.3 Memory and Register Snapshots

ER SNAP$ requests the Exec to print a snapshot of memory and/or registers. It uses a 3-word packet, shown in Figure 12-1, on the following page. These are the fields in the SNAP$ ER packet:

- The snapshot-id is a 6-character Fieldata string. This character string appears on the header line of the snapshot.

- XAR is a 3 bit field. For each bit set in this field, the Exec will snap the corresponding

registers. Bit 0 (35) requests a snapshot of the X registers; bit 1 (34), the A registers; bit 2 (33), the R registers.

- The word-count tells how many words to snap.
- The start-address is the address of the first word to snap.
- Former-A0-contents is a place to save A0 to before setting up to call the ER (see example code below).

A word about the word count and start address: the word count may be zero, but the start address *must* be a valid address, whatever the word count. By careful choice of the XAR and word-count fields, you can request only registers or only memory to be snapped, or both.

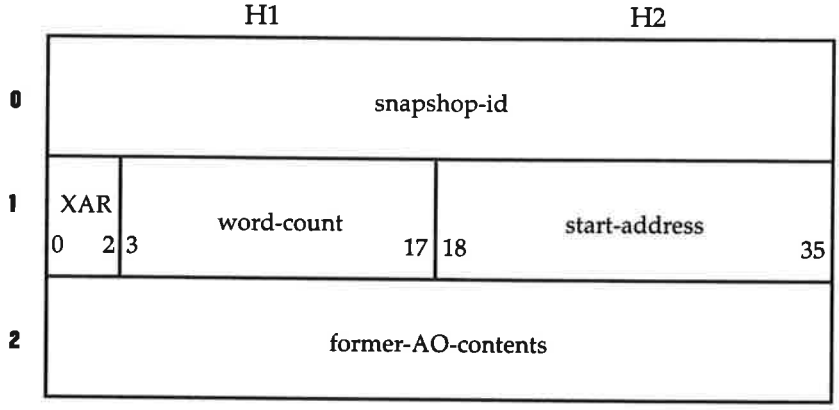

Figure 12-1. SNAP$ Packet Format

This is the calling sequence for ER SNAP$:

```
S           A0,<snap_pkt>+2
L,U         A0,<snap_pkt>
ER          SNAP$
```

Note that setting up the address of the packet in A0 destroys the contents of A0. In processing an ER SNAP$, the Exec always restores the contents of word 2 of the packet to A0. (Hint: always store A0 away in the packet if its contents are critical.)

Here is a code sample using SNAP$:

```
SNFORM      FORM        3,15,18          . FORM FOR WORD 1
SNAPKT      'BADRTN'                     . WORD 0
            SNFORM      6,27,TABLE       . REGS, WDS, ADDRESS
            $RES        1                . A0 SAVE

            S           A0,SNAPKT+2      . SAVE A0 OVER ER
            L,U         A0,SNAPKT        . ADDRESS OF PACKET
            ER          SNAP$            . ASK FOR SNAP
```

There is a much easier way to get a snap, and you don't have to allocate a packet elsewhere in the program. The system-defined procedure L$SNAP generates the 3 instructions shown, plus the packet and a jump around it. This allows you to code a snap virtually anywhere (except in a write-protected bank). Here is the format for using L$SNAP:

```
L$SNAP      <snap_id>,<xar>,<word_count>,<address>
```

Here is a sample call on L$SNAP which might replace the SNAP$ call above. Put this into your instruction code:

```
L$SNAP      'BADRTN',6,27,TABLE
```

It takes a lot fewer lines to code, doesn't it? This sample snaps only the registers (X, A, and R), and no memory locations:

```
L$SNAP      'BADEOF',7,0,0
```

The "0,0" construct tells SNAP$ not to snapshot any memory, or you can use 0,$. If <word_count> is non-zero, you must have a valid, non-zero <address>.

Make sure that a $FDATA directive is in effect when you use L$SNAP, so that a Fieldata string is passed to the ER. If you accidentally generate a 6-character ASCII string (2 words), the Exec will see bad parameters in the second word of the packet. The Exec will terminate your program in error. Chapter 16 will show a way to build a "shell" around L$SNAP for use in code where a $ASCII directive is in effect.

L$SNAP generates 6 words: a store of A0, a L,U, the ER, a J $+4 and the three-word packet. You can use this knowledge to advantage when you want to snap a table the address of which is in an index register. Suppose the pointer to the table is in X4. Here is the code to snap a 50-word entry beginning with the address in X4:

```
S,H2        X4,$+6      . ADDRESS OF ENTRY
L$SNAP      'BADEOF',7,50,$ . REQUEST SNAP
```

H2 of $+6 is the address field of the ER packet. The sequence of instructions generated by L$SNAP should be stable for a long time, so you can count on this working.

STYLE: There are several situations in which you can use SNAP$:

- In the middle of a set of complicated calculations, to check intermediate results. ("Complicated calculations": something like Z=(X-2)*27+(X-3)//4.) Typically, you will want to dump only the A registers (or wherever you are doing the calculations).

- In an error situation, to show details which will help diagnose the cause. If you can print an error message before doing the snap, better still. But printing the error message will destroy the contents of A0, so you may need to do the snap first. While the information in the snap will also be in a PMD, doing the snap will focus attention on the relevant information. Isolating the bad table entry or ER packet will save you time.

- Just before an ER, to make sure that your program is passing the correct information to the Exec.

- Just after an ER, to verify what the Exec is returning. If you are not sure that your program is testing a return status correctly, snap the area with the return status. That may be the packet in memory or the A registers.

- Just before or after a subroutine call, for reasons similar to those mentioned for ERs.
- To say "I got here". If you are not sure which branch your program is taking at a key decision point, put snaps in line with both paths. This gives a printed clue about what the program did.

Don't put a snap in the middle of a loop which will be executed thousands of times, unless you are prepared to look at the output each of one! At least, don't print the resulting print file; just scan it from your terminal. Save a tree.

Exercises

E12-1. Code a SNAP$ packet to display 15 words beginning at the label TEMP99 and the A registers. (Hint: you may need to code a $FORM first.) Code the instructions needed to perform the snapshot.

E12-2. Code an L$SNAP reference to display the X and A registers only (no memory to be displayed).

Chapter 13: Symbiont ERs

13.1 Symbiont ERs

In ordinary English usage, "symbiont" is defined as, among other things, a parasite or a biological partner. In OS 1100, though, the symbionts are a group of programs within the Exec which get runs into and out of the system. They also transfer runstream data to programs and accumulate printed output until a run terminates. (See Figure 13-1.) The diagram shows the symbiont files for batch runs; in the case of demand, READ$ and PRINT$ are the demand terminal. Of course, if the demand user has @BRKPTed PRINT$, PRINT$ is a disk file. And when a demand user @ADDs a file or element, READ$ information comes from the @ADDed file or element.

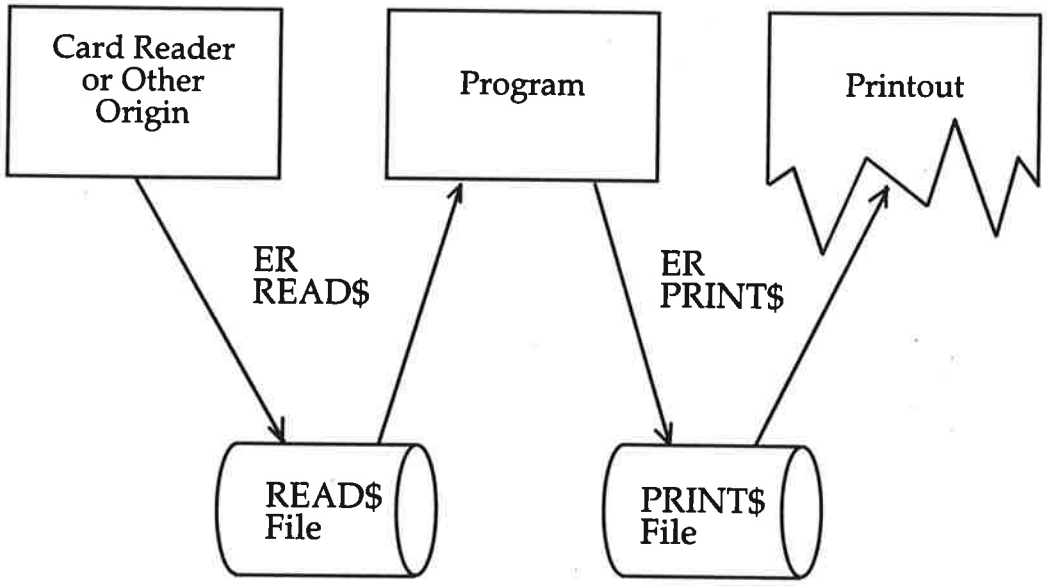

Figure 13-1. Symbiont Files and ERs

STYLE: The author understands that there is a long-term shift from Fieldata to ASCII. It is a good idea to code new programs to use as much ASCII as possible. But the transition will take years to complete and many ERs still require Fieldata parameters. As a result, you should be familiar with both character sets and the symbiont ERs for both.

13.1.1 Read Runstream Data In Fieldata - ER READ$

The first symbiont ER most programmers learn is READ$. Here is the general form for using READ$:

```
        L       A0,(<eof_return>,<buffer>)
        ER      READ$
```

There are two parameters passed in the two halves of A0. (And this ER always uses A0. Almost all ERs pass parameters via A0 and/or A1.)

- The parameter in H2 of A0, <buffer>, is required; this address tells the Exec where to transfer the image from the runstream.

The buffer should be 14 words long. (The maximum 80 Fieldata characters divided by 6 characters per word is 14.) The Exec assumes the buffer is long enough to accept a full 80-character image. Even if your program expects shorter images, you should allocate the full 14 words in case of an error in the input stream.

- H1 of A0 contains an optional end-of-file return address. The Exec will transfer control to this address when the runstream data has been exhausted. An @EOF or other control image marks the end of runstream data.

You may omit the end-of-file return address, but only if your code knows when it has read the last runstream image. If your program attempts to read past an end of file, and you have not specified an end-of-file return address, the Exec will terminate your program in error.

Here is a sample use of READ$:

```
READIM  $RES    14          . READ$ BUFFER
        L       A0,(PH1100,READIM)  . READ$ PARAM
        ER      READ$       .
.   (PROCESS IMAGE HERE)
.
.
PH1100  .
.   (PROCESS END OF FILE HERE)
```

The code at PH1100 might print an error message such as "premature end of file", then error exit. Chapter 16 will offer a way to make printing such a canned message easier—an ERRMSG proc.

Upon return from the ER (normal or EOF return), H2 of A0 will contain the number of words of information transferred to the buffer. (The last word returned by the Exec is blank filled, but the remainder of the buffer is unchanged. The remainder may contain data from a previous read.) H1 will contain bit settings including these:

- Bit 0 (35) is set (hint: use JN or JP instruction to test) when a control statement is next in the runstream and it cannot be bypassed. This bit is set on EOF return. Further hint: don't attempt to read any more.

- Bit 1 (34) is set when reading from an @ADD file (or element).

- Bit 2 (33) is set when reading from a file that was @ADDed using an E option and the file has been exhausted. Set on EOF return.

The other bits involve INFOR and CLIST$, which are beyond the scope of this book. This is one way to test for a bit other than 0:

```
TOP         A0,(+ 1*/34) . BIT 1 SET?
J           NO$ADD       .  NO,NO$ADD  YES,
```

(TOP, the Test Odd Parity instruction, is covered in Chapter 11. The */ operator will be described in Chapter 17.) This sequence jumps when bit 1 (34) is clear.

By handling the EOF returns correctly, you can put multiple files in the runstream, as in this example:

```
@XQT        MYPROG
27
11
22
@EOF
18
19
99
@EOF
@EOF
```

Here's the technique:

- Set up a flag cell, initialized to zero.
- Upon EOF return from READ$, if the flag is non-zero, don't read any more.
- If the flag is zero, set the flag, and go back and try to read more.
- If the new read is successful, clear the flag cell.

The only kink in the algorithm is that you need to test for bit 0 when you get an EOF return—if it is set, of course, stop reading immediately! You might get bit 0 set if somebody other than you uses the program, and instead of the two @EOFs at the end, uses an @BRKPT or something besides @EOFs.

You can replace the two instruction lines (load and ER) with one procedure call, R$EAD. R$EAD is defined by the system and available to any MASM user. Procedures are described in Chapter 16, but this revision of an earlier example shows the basics for R$EAD:

```
READIM      $RES        14          . READ$ BUFFER
            R$EAD       (PH1100,READIM) . READ$ PARAM
.    (PROCESS IMAGE HERE)
.
.
PH1100      .
.    (PROCESS END OF FILE HERE)
```

The R$EAD procedure generates the load and ER instructions just as if you had hard-coded them

yourself. It inserts values from the operand field into the instructions. Use or make shortcuts like R$EAD whenever possible; they will save coding time.

If the @ADD file or element happens to be in ASCII or mixed ASCII and Fieldata, the Exec will translate it to Fieldata as necessary to satisfy the READ$ request. In other words, READ$ always returns a Fieldata image, regardless of which character code the runstream is in.

If the buffer address is outside your program (including zero), the Exec will terminate your program in error. If the EOF return address is outside your program, the Exec will give your program a guard mode contingency when it reaches an end-of-file.

13.1.2 Fieldata Printing: PRINT$

The flip side of the symbionts is printing. PRINT$ transfers one Fieldata image from the buffer you specify to the PRINT$ file. The ER needs three parameters in A0, dividing A0 into 12, 6, and 18 bits:

```
PF          $FORM       12,6,18                    .
            L           A0,(PF <spacing>,<words>,<buffer>)  .
            ER          PRINT$                     .
```

Of course, you need define the form, PF, only once (probably near the beginning of your program). The name "PF" is not required, but suggested for consistency with other people's code. These are the three fields of the parameter word:

- <Spacing> tells how many lines to space before printing this image. If <spacing> is greater than the number of lines remaining on the page, the line is printed at the top of the next page.
- <Words> tells how many words of Fieldata information to transfer. The maximum is 22 words, that is, 132 characters divided by 6 characters per word equals 22.
- <Buffer> is the location of the image.

The spacing parameter can be any value from 0 to 07777 (all 12 bits set). A value of zero causes the printer to print this image over whatever was printed on this line before. While overprinting can be used for special effects, building up a print line by printing one field in each image is not considered sporting. Besides, overprinting slows down the printer. A value of 07777 is a guaranteed page skip.

The spacing value and page ejects may be handled differently for demand runs, depending on the terminal or communications front end processor.

Here is a sample use of PRINT$:

```
            $FDATA                                 .
BUFFER      'THIS IS A SAMPLE MESSAGE'             .
            L           A0,(PF 1,4,BUFFER)         . PRINT$ PARAM
            ER          PRINT$                     . ASK FOR PRINT
```

Just as for READ$, the system defines a PRINT$ call procedure which combines the parameter load and the ER on one line. Here is an example use of it:

```
            $FDATA
BUFFER      'THIS IS A SAMPLE MESSAGE'      . PRINT IMAGE
BUFFL       EQU         $-BUFFER            . BUFFER LENGTH
            P$RINT      (PF 1,BUFFL,BUFFER) . ASK FOR PRINT
```

Actually, this sample also shows another shortcut. The technique of $-BUFFER in the next line after BUFFER saves you three kinds of trouble:

- You don't have to count the characters in the message, ever.
- If you change the text of the message, the word count changes automatically.
- You might make a mistake in the word count; MASM never will, if you code this construct correctly.

Note that the character string in BUFFER will be left justified and space filled out to a full-word boundary.

About the only time you need to use ER PRINT$ without the P$RINT proc is when your program must construct the image and the parameter word for it dynamically. Chapter 11 has an example showing the construction of a PRINT$ parameter.

If your PRINT$ parameter tries to transfer too many words, or specifies a buffer outside your program, the Exec will terminate your program in error.

13.1.3 ASCII Variations

Using the ASCII symbiont ERs, you can transfer images in ASCII to or from the symbiont files PRINT$ and READ$. (The symbiont files and their names stay the same, regardless of which ERs you use to access them.)

To read ASCII images from the input stream, use ER AREAD$. The parameters are exactly the same as for READ$:

```
            L           A0,(<eof_return>,<buffer>)  .
            ER          AREAD$                      .
```

The main difference is that you need to allocate a 20-word buffer, instead of 14 words (80 characters divided by 4 characters equals 20 words). And, once again, if the runstream data is in Fieldata or mixed Fieldata/ASCII, the Exec will translate it as necessary to deliver ASCII to your buffer. Coding for AREAD$ is the same as for READ$, so you can refer to the READ$ examples above, just changing the ER name to AREAD$.

The system-defined procedure to call AREAD$ with a parameter word is A$READ. The use of A$READ is just like R$EAD:

```
            A$READ      (<eof_return>,<buffer>)  .
```

To transfer ASCII images to PRINT$, use ER APRINT$:

```
L       A0,(PF <spacing>,<words>,<buffer>)
ER      APRINT$
```

(This assumes you have defined PF elsewhere in your program.) Once again, the parameters are exactly the same as for PRINT$. The maximum number of words is now 33 (132 characters divided by 4 words per character equals 33 words). You can mix Fieldata and ASCII in the print file; the Exec will translate as necessary for the output device, printer or terminal.

The system-defined procedure for calling APRINT$ is A$PRINT. Here is the format:

```
A$PRINT   (PF <spacing>,<words>,<buffer>)
```

Are you beginning to see a pattern in the names for the ERs and their calling procedures? A later part of this chapter will have a table of the basic (Fieldata) ER names, their ASCII counterparts, and the calling procedures.

13.2 Alternate File Symbiont I/O

13.2.1 Reading From An Alternate File

The next group of ERs lets data files you have built stand in for the READ$ or PRINT$ files built by the symbionts. (You may not need to use alternate file I/O in your first exercises; you may skip this section on a first reading.) Since the functions are almost identical, the parameters are very similar. There are two main differences:

- The parameters include the name of the data file to read or write.
- Since there are now three words of parameters, you must put the parameters in memory; A0 must contain a pointer to the parameter words.

See Figures 13-2 and 13-3 for the parameter word formats (following pages).

The alternate file counterpart of READ$ is READA$ (read alternate). The packet (1100 terminology for an ER's parameter list) begins with the EOF return address and a buffer address. These have exactly the same meaning as for READ$. Note that ER packets always begin numbering words with 0 rather than 1.

Words 1 and 2 contain the Fieldata name of a data file. You must have previously built this file, it must contain SDF data, and it should be assigned to your run. You can build an SDF data file with CTS or EDIT-1100.

This is how to call ER READA$:

```
L,U     A0,<read_pkt>
ER      READA$
```

The L,U places in A0 the address of <read_pkt>, a packet in the form shown in figure 13-2. (Note the difference between this and READ$, which accepts parameters directly from A0.) The system defines a procedure to call this ER. Use it like this:

```
            R$EADA      <read_pkt> .
```

Here is a code fragment using R$EADA:

```
            $FDATA
IMAGE1      $RES        14          . READA$ BUFFER
RDAPKT      +           EOFRTN,IMAGE1 . READA$ PARAM
            'MYDATA                 . ALT FILE NAME
            R$EADA      RDAPKT      . GET 1 IMAGE
.    (PROCESS IMAGE HERE)
.
.
EOFRTN      .
.    (PROCESS END OF FILE HERE)
```

The $FDATA directive at the start of this fragment has only one purpose: to ensure that the file name in RDAPKT is in Fieldata. The only way the Exec knows to transfer Fieldata into your buffer is by the name of the ER you call.

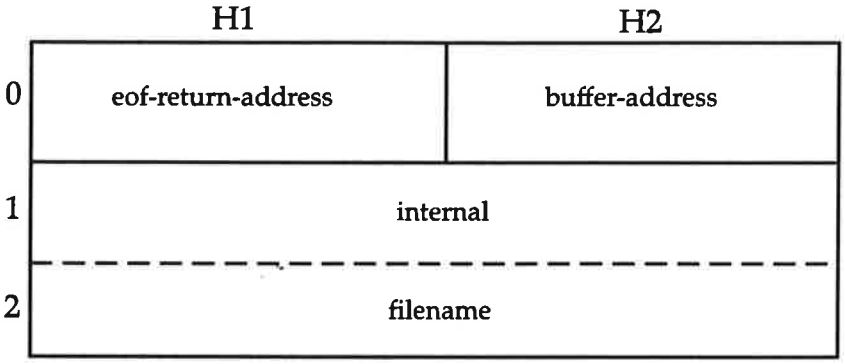

Figure 13-2. READA$

There is no label on the line which contains the file name, because we don't need the address of those two words. If the file name needed to be set up at execution time, we could approach it two ways. First:

```
RDAPKT      +           EOFRTN,IMAGE1 . READA$ PARAM
RDAFNM      'MYDATA       '           . ALT FILE NAME
```

That is, simply put a label on the same line. Or:

```
RDAPKT      +           EOFRTN,IMAGE1 . EOF, BUFFER
            'MYDATA       '           . ALT FILE NAME
RDAFNM      $EQUF       RDAPKT+1      .FIELD DEFINITION
```

For this simple a packet, the choice is yours. For more complex packets, you should *always* define the packet with a single label at the start and define the fields with $EQUFs at the end. The $EQUFs should specify offsets from the beginning label of the packet data. Those offsets are easy to check against the ER documentation. Putting all the field definitions together makes the code easier to read.

The file you tell the ER to read from should be assigned to your run; the Exec will not do the assignment. If the file has a qualifier different from your run, you will also need to attach an @USE name to it, creating an "internal" name. The examples above specify the file name MYDATA in the

packets. If you want to read the file RESEARCH*PROJECT27 with the packets above, use this ECL before executing the program:

```
@ASG,A      RESEARCH*PROJECT27.
@USE        MYDATA.,RESEARCH*PROJECT27.
```

or

```
@USE        MYDATA.,RESEARCH*PROJECT27.
@ASG,A      MYDATA.
```

Either way will work.

13.2.2 Printing To An Alternate File

The print counterpart of READA$ is PRNTA$, print alternate. The packet format is shown in Figure 13-3. Word 0 contains the same information as the A0 you supply for PRINT$. These three fields, spacing, word count and buffer address, are handled the same as for PRINT$. Words 1 and 2 contain an internal file name, in Fieldata. ("Internal" means you have attached an @USE name, as necessary.)

The calling sequence for PRINTA$ is:

```
L,U         A0,<print_pkt>      .
ER          PRNTA$              .
```

The system defines a procedure for calling PRNTA$:

```
P$RNTA          <print_pkt>     .
```

The file specified in words 1 and 2 does not have to be assigned to the run before your first ER. The action the Exec takes with your file depends on whether the file was assigned before the first ER or not:

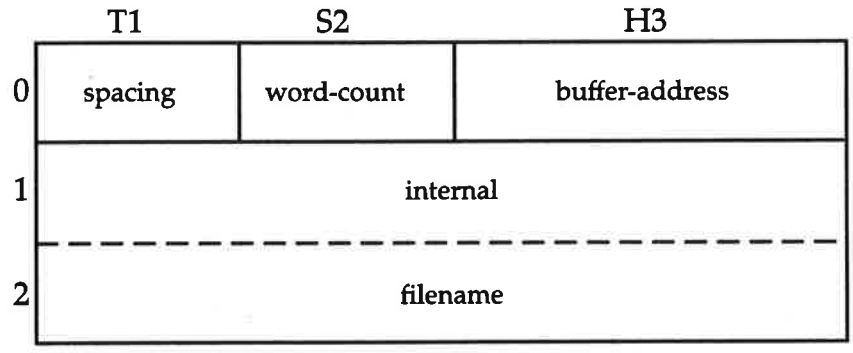

Figure 13-3. PRNTA$ Packet Format

- If the file was not assigned before the first ER PRNTA$, the Exec will automatically assign a file for you. When your run closes the file by either an @BRKPT or @FIN (end of run), the Exec will queue the file for output for you. (Where the Exec queues the file to depends on where your run came from, but in most cases it will be PR, the onsite printer

group.) This automatic action applies only to batch runs. The Exec will not perform automatic assignment for demand runs.

- If the file was assigned to your run before the first ER PRNTA$, the Exec will take no automatic action. It is up to your run to close the file explicitly with an @BRKPT. Your run then has the option of @SYMing the file to an output device, as desired.

Closing an alternate file requires a little different @BRKPT statement than used with PRINT$. Instead of PRINT$, specify the name of your alternate file:

```
@BRKPT      INTERNAL
```

The actual sequence to create an alternate print file is a little tricky. Here it is for review:

- `@ASG,UP  QUAL*FN.,F`
- `@USE     INTERNAL.,QUAL*FN.`
- Your program does an ER PRNTA$ for each image.
- `@BRKPT   INTERNAL`
- `@FREE    INTERNAL.`
- `@SYM     QUAL*FN.,,PR`

This order is very important. The Exec writes the SDF end-of-file image to the file when you @BRKPT it, and the file must be assigned to your run in order to do that. If you @FREE it before the @BRKPT, it cannot properly close the file.

The @BRKPT must *not* have a period after the file name.

13.3 ASCII Alternate Symbiont File I/O

Finally, the alternate file ERs just described have ASCII counterparts, too. They transfer ASCII images from or to user-defined files. The packet formats are identical to those just described. The file name field is still Fieldata—don't let that throw you. The file handling is also the same.

The ASCII read alternate ER is AREADA$. The corresponding call procedure is A$READA. The ASCII print alternate ER is APRNTA$ (note the missing I). Its system-defined call procedure is A$PRNTA. You might want to refer to your Quick-Reference Guide to check the format of these procedures. (Hint: use them the same way as R$EADA and P$RINTA, respectively.)

13.4 Symbiont ER Name Review

There is a definite pattern to the names of the symbiont ERs. This table should help you see the pattern:

	Fieldata	Proc	ASCII	Proc
Read	READ$	R$EAD	AREAD$	A$READ
Read Alt	READA$	R$EADA	AREADA$	A$READA
Print	PRINT$	P$RINT	APRINT$	A$PRINT
Print Alt	PRNTA$	P$RNTA	APRNTA$	A$PRNTA

The order of each name is [ASCII]<function>[alternate]. An A is added at the start of the name for ASCII, and on the end for "alternate". The name of each procedure is created from the ER it calls by moving the $ (dollar sign) from the end to the second character.

13.5 Print Control Functions

(You can skip this section on a first reading of the book, if you wish.) Using these ERs, you can control a number of aspects of the way your run is printed:

- Control the number of vertical lines per inch, either 6 or 8.
- Start printing on a particular line of a new page.
- Print a heading on each page, with or without page number and date.
- Request special forms, with a message displayed at the time the printer is printing the printfile.
- Set page length and top and bottom margins.
- Set line width to something other than the default of 132 characters.

These ERs place control records in your PRINT$ file; they are interpreted at print time. (Have you ever tried to ask the operator to mount special forms, but could not get the forms mounted at just the right time? Read on.)

Like the other symbiont ERs discussed so far, there are Fieldata and ASCII variations, and PRINT$ and alternate file variations. The Fieldata/PRINT$ version is PRTCN$ (mnemonically, think of it as "Print CoNtrol"). Here is the call format:

```
L       A0,(<words>,<buffer>)  .
ER      PRTCN$                 .
```

The parameter in A0 consists of a buffer address in the right half and the word length of the buffer in the left half. The contents of the buffer are an image in Fieldata consisting of a command letter, a comma and parameters which vary according to the command letter. For example, here is the format for a margin change:

```
M,<length>,<top>,<bottom>,<lpi>
```

where:

- <length> is the page length in lines,

- <top> is the top margin in lines,
- <bottom> is the bottom margin in lines, and
- <lpi> is the number of lines per inch (limited to 6 or 8 by most printers).

If any of these parameters is omitted (at least for the margin command), the system will leave that value as it is. This allows you to change only one parameter, leaving the rest alone. You can also use a parameter value of * (asterisk) to request the system standard value. (You may need to ask your Exec support group for the standard values, or you may be able to count them on a sample printout.) Here is a margin parameter and the corresponding PRTCN$ call:

```
            $FDATA
MRGBUF      'M,66,2,2,6'
MRGLEN      $EQU        $-MRGBUF    .
.
            L           A0,(MRGLEN,MRGBUF)  . PARAMETERS
            ER          PRTCN$      . DO THE ER
```

This call will set the page length at 66 lines at 6 lines per inch (that is, 11 inches), with top and bottom margins of 2 lines each.

The next command letter is B, for controlling lines per inch and other things beyond the scope of this book. The format of the B command is:

```
B,<lpi>
```

where <lpi> is the number of lines per inch, 6 or 8. (By the way, all bets are off for laser or other non-impact printers. They may ignore or otherwise alter your print density request.)

The L command tells the printer to start printing at a requested line number:

```
L,<nn>
```

where <nn> is a line number, one less than where you want printing to begin on a page. If the next print request has a line count of 1, it will print on line <nn>.

The heading command, H, is more fun:

```
H,<option>,<page>,<text>
```

where:

- <option> is an option letter, either N or X. This field is most often used to turn off a heading, that is, 'H,N' with no other parameters. The X option suppresses printing of the page number and date.
- <page> is a page number. Using this field, you can start page numbering at an arbitrary number at the beginning of a file, or start it over in the middle of a file.

- <text> is the heading to be printed at the top of each page. It may be no more than 96 characters, and it may not contain any periods.

The heading is printed within the top margin of each page. If there are not enough lines to print the heading, the system will not print the heading and it will not give an error message. Here are some sample heading commands:

```
HDR1        'H,,,TAPE REEL NUMBER UTILITY'
HDR2        'H,X,,VERSION 2'
HDRN        'H,N'
```

HDR1 makes the system put a heading of TAPE REEL NUMBER UTILITY plus page number and date on each page. Page numbering will continue with one greater than the previous page. HDR2 will put a heading of VERSION 2 on each page, but there will be no page number or date. HDRN turns off the headings completely.

The command letter S requests a special form:

```
S,<request>
```

where <request> may be up to 48 characters long, and may not contain a period. Here is a sample special forms request:

```
FRTBIL      'S,PLEASE MOUNT FREIGHT BILLS'  .
```

Note that if you take no other action, the system will return to standard forms when your print file has been exhausted.

One last command letter is W. If your system has a non-standard printer capable of greater than 132-character lines, you will need this command to use the full carriage width. Here is the format:

```
W,<words>
```

where <words> is maximum number of words to be accepted per print line at 6 characters per word. The default is 22. This parameter affects ERs PRINT$, SYMB$, APRINT$, PRINTA$, or the print parts of TREAD$ and ATREAD$.

You can combine multiple commands in one buffer. Precede additional commands immediately by a period, like this:

```
PRTC1       'H,,,GENERAL LEDGER .S,MOUNT G/L FORMS.L,2'
```

Note that the first command in the buffer must not be preceded by a period.

There are less frequently-used commands available. You may want to check your PRM to see the others. (Look in UP-4144.32, Section 5.4, and UP-4144.38, Section 16.)

All of the commands just described (M, B, L, H, S, and W) are supported by the other print control ERs. The ASCII variation is APRTCN$:

```
L       A0,(<words>,<buffer>)  .
ER      APRTCN$    .
```

That is exactly like PRTCN$, except that the image must be in ASCII. The word count must reflect the fact that there are only 4 characters per word in the buffer.

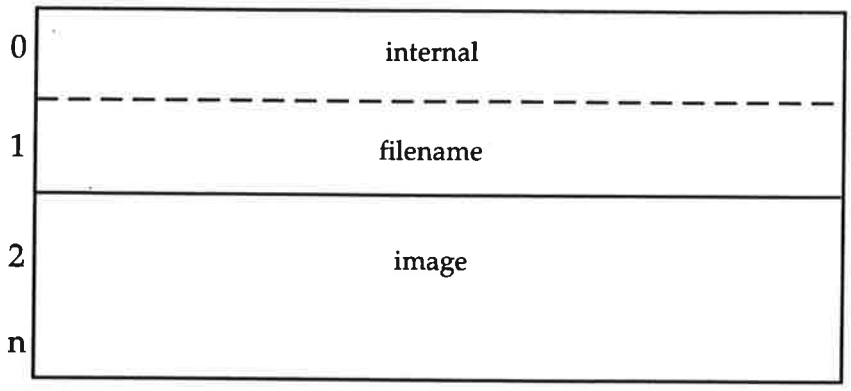

Figure 13-4. PRTCA$. APRTCA$ Packet Format

The buffer format for PRTCA$ and APRTCA$ is shown in Figure 13-4. These two ERs apply print changes to a named alternate print file. The first two words of the buffer contain the Fieldata internal filename of the file to which the change applies. (Note that the filename is in Fieldata for both ERs.) The remainder of the buffer is the print control image itself: Fieldata for PRTCA$, and ASCII for APRTCA$.

Here is a sample sequence for APRTCA$:

```
            $FDATA
PRTCTL      'MY-FILE      '          . FD FILE NAME
            $ASCII
            'H,,,MY-ALT HEADING'
PCLLEN      $EQU        $-PRTCTL     . IMAGE LENGTH
.
            L           A0,(PCLLEN,PRTCTL)  . PARAMETERS
            ER          APRTCA$      . DO THE ER
```

13.5.1 Print Control Via ECL

One other note: you may not need to use MASM to effect these print changes. The @HDG control statement will accept any of these commands. Just put a period after the actual heading text, and follow it immediately by the first command letter. You can also combine commands the same way. Here is an example:

```
@HDG  **MASM LISTING FOR FILE**.M,66,6,3.S,MOUNT 2-PART FORMS
```

This image would request 2-part forms, set the margins at 6 top and 3 bottom on a 66-line page (presumably at 6 lines per inch) and print the heading **MASM LISTING FOR FILE** on the top of each page.

Exercises

(Hint: You may need to use SNAP$ or L$SNAP to complete these exercises. If your code works the first time, insert a snap of the returned A0 from each READ$ or AREAD$.)

E13-1. Code up a little program to read Fieldata images from the runstream and echo them to the printer. Make sure your test data has at least a few images 80 characters wide.

E13-2. Copy the program from the previous exercise and make a version which works in ASCII. Make sure it echoes the data as accurately as before, without truncation.

E13-3. Copy one of the first two programs, and make it copy the data to an alternate print file. Read the file after execution using your favorite editor. SYM it to a printer and look at the results.

E13-4. Modify the program from E13-3 to print the alternate file at 8 lines per inch (6 if your default is 8) with top and bottom margins of 4 lines each.

Chapter 14: Subroutines

14.1 Types of Subroutine Entries (Labels)

The subroutine instructions to be discussed can use two kinds of labels, local and external, as their targets.

- A local label is defined within the same element as the reference to it (that is, an instruction line).
- An external label is defined in a different element and is externalized (in MASM, at least) by an asterisk, like this:

```
CALL15*     . <purpose of subroutine>
```

Of course, other language processors can create externalized labels, by means of statements or clauses like ENTRY, EXPORTED, or the like.

STYLE: Part of the job with MASM is checking for undefined entry point messages from the collector. If you misspell the name of a subroutine and that subroutine is defined in a different element, MASM will not catch it but the collector will. Undefined entry points are included in the collector's error count on its summary line at the end of its print output, *viz.*, "END MAP, ERRORS: 0". Don't let this slide by unless you like surprises.

14.2 EX: A "Mini-Subroutine"

The Execute Remote instruction (EX) is only used to execute other instructions. You can think of it as a way of calling a very small subroutine, consisting of only one instruction. Here is an example use of EX:

```
TAB027      L,S2        A3,BUFF,X5 .
.
            EX          TAB027     .
```

Chapter 21 will show how to use the EX instruction for a less trivial purpose than this example. In general, you can modify u with an index register to choose which instruction gets executed; that is the power of the EX instruction.

14.3 SLJ: "Sludge"

The most elementary subroutine call instruction is SLJ, Store Location and Jump. It captures the current value of the program counter, adds 1 to it and stores it in H2 of U, then jumps to U+1. Here is an example use of SLJ:

```
                .
                .
                .
                SLJ         SUB1        . CALL SUB1
                NI
                .
                .
                .
SUB1            J           $-$         . RETURN CODE
                <SLJ ACTUALLY JUMPS HERE> .
                .
                <WORK OF SUB1>
                .
                J           SUB1        . EFFECT RETURN
```

How do you get back from a subroutine called by SLJ? Notice the J SUB1 at the end of the example; it transfers control to the jump instruction at SUB1, which in turn returns control to the instruction following the SLJ.

There is only one advantage to using SLJ over other subroutine call instructions. It does not change any registers; this could be a big help in a register-tight sequence. There are critical disadvantages to using SLJ:

- Most importantly, it changes a word in instruction code. If you are creating reentrant code, you will probably want to put your instruction code in read-only banks, and that means no SLJs. Worse, if multiple routines call an SLJ routine, the return address of one or more activities gets lost. (Hint: *never* use SLJ with multiple activities.)

- The second problem is that creating multiple return points (see LMJ, next) with SLJ requires a register, negating one of the advantages of the instruction.

- Finally, there are ways to allow instructions to exist at addresses over 65K, but in that case, you can't return from an SLJ as shown in the example; the h or i bits are set. You have to return on an index register anyway.

SLJ is shown here so you can read already-existing code; you should understand the limitations of the instruction and use cleaner instructions for subroutine calls. SLJ is not supported in extended mode.

14.4 LMJ - Load Modifier and Jump

LMJ is the main subroutine call instruction you will use for local subroutines and subroutines in programs with only one I bank. It captures the current program counter (denoted "P" in the instruction summary chart), adds one to it, places that value in X_a, then jumps to U. Return is a simple indexed jump, as in this code fragment:

```
                .
                .
                .
                LMJ         X11,SEARCH1  . CALL SEARCH1
                NI          .
                .
                .
SEARCH1         .
                (CONTROL STARTS HERE FOR SEARCH1)
                .
                J           0,X11        . RETURN TO CALLER
```

When control is passed to SEARCH1, X11 contains the address of the instruction following the LMJ. All that's needed to get back to the calling point is the J 0,X11 at the end of SUB1.

A subroutine may signal to the caller by means of multiple returns, as in this example:

```
                .
                .
                .
                LMJ         X11,SEARCH1  . CALL SEARCH1
                J           ERRPATH1     . NO FIND
                J           ERRPATH2     . INVALID KEY
                NI                       . NORMAL RETURN
                .
                .
SEARCH1         .
                (CONTROL STARTS HERE FOR SEARCH1)
                .
                J           2,X11        . NORMAL RETURN TO CALLER
                .
                J           1,X11        . INVALID KEY
                .
                J           0,X11        . NO FIND
```

In this example, 0,X11 and 1,X11 become error paths for the SEARCH1 subroutine. Passing a return status in this way does not use up a register.

STYLE: If you use multiple return points, it is important to route all the code in that subroutinewhich will use a given type of return through the same exit point. Changing the return conventions for a subroutine is a lot easier if you limit the number of places you return from. Likewise, if you need to insert a memory snap just before a subroutine return, it is a lot easier if there is only one return of each type. Please notice the distinction between these two returns:

RETURN TYPE A:

```
          TE,U        A5,KEY1          . IS THIS THE RIGHT ONE?
          J           0,X10            . NO,RETURN  YES,
```

RETURN TYPE B:

```
          TE,U        A5,KEY1          . IS THIS THE RIGHT ONE?
          J           SRCH10           . NO,RETURN  YES,
          .
          J           SRCH10           . MAKE RETURN
          .
SRCH10    . THE LAST KEY WAS NOT RIGHT .  RETURN FAILURE.
          J           0,X10            . RETURN TO CALLER
```

Can you see the difference? The first way, inserting a memory snap or other debug call is impossible without minor surgery. You might miss this return jump because it is "buried" under the TE instruction. The second way makes the return much more visible. It's easier to find and easier to change. The label gives us a place to hang an explanatory comment, or even a whole paragraph of comments. Subtleties like this are what make a program a joy to maintain, or a nightmare to debug!

Chapter 19 has suggestions for deciding which registers to use for calling subroutines and which to use for pointing to tables.

When an EX (Execute Remote) is used to execute an LMJ, the address captured in X_a is the address of the EX instruction, not the address of the LMJ. That feature can be used to advantage when manipulating characters (see Chapter 21).

14.5 LIJ, LDJ, and LBJ

These three instructions base new banks. They are also used to call subroutines. *The detailed descriptions are beyond the scope of this text; the brief descriptions below are included in case you are called on to read existing code using these instructions.*

- Load I-bank Base and Jump - Switches guard mode and basing registers to point to the I bank the BDI of which is in XI(X_a). Captures the previous bank's BDI and program address in XI(X_a) and XM(X_a), respectively. Jumps to U.
- Load D-bank Base and Jump - Switches guard mode and basing registers to point to the D bank the BDI of which is in XI(X_a). Captures the previous bank's BDI and program address in XI(X_a) and XM(X_a), respectively. Jumps to U.
- Load Bank and Jump - LBJ is the general case of LIJ and LDJ. LBJ accepts a bank descriptor register number in bits 1 and 2 of X_a, where the BDRs used by LIJ and LDJ are implicit.

The BDI and program address captured in X_a are structured so that the called subroutine can effect a return by a simple LxJ Xn,0,Xn. The saved BDI in XI(Xn) tells the CPU which bank to base; the modifier portion contains the return address.

LDJ is used less often for subroutine calls than basing D banks. The U address (jump destination) does not have to be in the bank which is being based.

If a more detailed description is required, please refer to the Processor and Storage reference for your system, or the AIM.

14.6 Register Save and Restore Strategies

Internal subroutines in your programs will need to save and restore registers. The number of registers you have to preserve depends on the length and other characteristics of your subroutines. For example, you probably want to keep A0, A1, and possibly A2 as *volatile*, meaning that code which uses them will keep values there for a few lines at most. Keep more permanent values, like tallies in a loop algorithm, in higher-numbered registers.

Here are some suggestions:

- Avoid using registers the caller expects to be preserved.
- Save those critical registers in other registers you can afford to wipe out. If you use this technique, make sure you are *symmetric*, that is, store A5 into A10 at the beginning of the routine, then load A5 from A10 near the end. Loading A10 from A5, then loading A5 from A10, is less obvious. You want omissions of register restores to stand out clearly.
- Save critical registers singly in memory locations. (If the program has only one activity, this is easy to work out. Multiple activities require much more care.)

14.7 Save and Restore Register Sets - SRS, LRS

If a subroutine is long and involved, or called by Cobol or other high-level language, you will need to take more wholesale measures. This is one time-honored way to solve the problem:

```
S       X9,SAVE+0
S       X10,SAVE+1
S       X11,SAVE+2
DS      A1,SAVE+3
DS      A3,SAVE+5
DS      A5,SAVE+7
```

(Recall that there is no instruction to store two X registers at once.) This sequence saves X9 through X11 and A1 through A6. Two instructions not available to assembler programmers on the old 1108 make this kind of work easier: SRS and LRS. Here is code using SRS to do the same job:

```
L       A0,(+ 3,X9,6,A1)    . LOAD SRS PARAMETER
SRS     A0,SAVE
```

That's a lot less work, isn't it? Here are more formal descriptions of SRS and LRS:

- Store Register Set (SRS) - transfers the contents of registers to U and following locations in memory, using a parameter word in Aa. There are two pairs of parameter fields expected

in A_a. Each pair is composed of a register count and a register address, in that order, on quarter-word boundaries.

- Load Register Set (LRS) - transfers data from U and following locations in memory to registers, using a parameter word in A_a. The parameter word is of the same format as for SRS.

The SRS example above specified storing 3 registers starting at X9 and 6 registers starting at A1. In practice, clarity, symmetry and attention to detail suggest using only a single memory location to hold the LRS/SRS parameter word for a subroutine, as in this example:

```
SRSPARAM    +           3,X9,6,A1          .

SUB027      .
            L           A0,SRSPARAM        . GET SRS PARAMETER
            SRS         A0,SAVTAB          . SAVE X9-X11,A1-A6
            .
            .
            .
            L           A0,SRSPARAM        .
            LRS         A0,SAVTAB          .
```

Using the same word for the SRS/LRS parameter makes dead certain that exactly the same registers will be saved and restored, and in the same order. In fact, you can also make parameters of the 3 and 6, and these in turn can contribute to allocation of memory in a table, in this fashion:

```
REGSAV      $EQUF       FIELD9+2,X5        . REG SAVE AREA
XCOUNT      $EQU        3                  . NBR X REGS TO SAVE
ACOUNT      $EQU        6                  . NBR A REGS TO SAVE
FIELD11     $EQUF       REGSAV+XCOUNT+ACOUNT,X5  .
            .
            .
            .
SRSPARAM    +           XCOUNT,X9,ACOUNT,A1  .
```

In this example, REGSAV's offset "floats" on FIELD9's (whatever it is), and FIELD11's floats on REGSAV's. FIELD11's offset will change whenever XCOUNT or ACOUNT change. Using this technique, you can adjust the number of registers saved as you code a routine, without manually adjusting the space required to save those registers.

Since LRS and SRS can move two contiguous strings of registers at one time, you can save all the X, all the A and all the R registers with one SRS instruction. As it happens, the X and A registers occupy contiguous GRS (General Register Set) addresses. X1 is at address 01, X11 at 013, A0 at 014, and A15 at 033. The R registers, on the other hand, begin at address 0101 (non-contiguous with X and A registers). Therefore, a parameter word of "+ 27,X1,15,R1" will save or restore all X, A, and R registers. (Be aware that there are four more A registers available above A15, and LRS/SRS can address them, too.)

Exercises

E14-1. Look at an application program you have written lately. Identify the phases in it (more than 1 and less than 10), and code a mainline sequence to call each phase in turn, as if you were going to rewrite the program in MASM. Use the LMJ instruction. Use multiple returns as necessary.

E14-2. Take one of the exercises you did for Chapter 13 (reading and printing) and make it into a subroutine. Code a short mainline program to call it, then run the resulting program.

E14-3. When you have done the previous exercise, change the program so it can respond to commands in the runstream. An image consisting only of the string 'ASCII' signals a switch to reading and printing ASCII. An image consisting only of the string 'FDATA' signals a switch to reading and printing Fieldata. Create one subroutine for each mode, ASCII or Fieldata. If either subroutine detects a "switch" image, it should do an alternate return to the mainline so it can call the other routine. If either routine detects an end of file condition, it should do another type of alternate return to the main program, which should exit. There should be only one ER EXIT$ instruction. (Hint: the mainline must have a common exit point.) Finally, put some lower case alphabetic data into your test runstream to make sure your program is reading ASCII when it is supposed to.

Chapter 15: Repeated Instructions

15.1 Block Transfer

Using the BT instruction, you can copy one area of memory to another. Or, you can fill an area of memory with a constant. Here it is:

- BT (Block Transfer) - copies the contents of X_x+u to X_a+u under j partial-word control, repeating as many times as specified in R1.

Here is an example to copy 12 words from TAB1 to TAB2:

```
TABSIZ    $EQU      100          . NUMBER OF ENTRIES TO RESERVE
TABENT    $EQU      2            . ENTRY SIZE
COUNT     +         12           . REPEAT COUNT
TAB1      $RES      TABSIZ*TABENT . TABLE 1
TAB2      $RES      TABSIZ*TABENT . TABLE 2
          L         R1,COUNT     . GET REPEAT COUNT
          L         X5,(1,TAB1)  . STARTING SOURCE REGISTER
          L         X1,(1,TAB2)  . STARTING DEST. REGISTER
          BT        X1,0,*X5     . COPY 'COUNT' WORDS
```

Note that the registers are named in reverse order on the instruction; that is, the destination register is first, then the source register. The asterisk on *X5 sets the h bit, causing the CPU to add the increment portion of X5 to its modifier portion after each source word is obtained. In that manner, the CPU gets sequential words of the source area. The h bit also controls the incrementation process on the destination index register (X1 in this case). If you don't want the destination address to change as the instruction is repeated, set XI of the source register to zero.

Note that you must always specify the repeat count in R1, the repeat count register; the CPU is hardwired that way. Also, be aware that the instruction changes the contents of the X registers as it goes. If you have registers already set to point to the beginnings of tables and you don't want to lose them, transfer the pointers to A0, A1, or A2, and use them. They are usable as index registers and will normally be volatile.

STYLE: Block transfers are among the most important places to put meaningful comments. Use terms such as "source" or "from" and "destination" or "to" for clarity. The line which sets up the source register *must* have a comment to that effect; likewise, the destination register.

Having seen this example, you can probably guess the ways you can vary a block transfer:

- Table fill - setting the source register's XI to zero causes the BT to copy the same source word over and over in the destination area.

- Alternate words - setting XI of either the source or destination to a non-zero value other than 1 causes the instruction to pick every second, every third, etc., word.

- Partial words - you can transfer only portions of the source words to the destination by using the partial-word modifier.

Here is an example of the way to fill a table with a constant, in this case 2:

```
TABSIZ      $EQU        100          .
TABLE       $RES        TABSIZ       .
            L,U         R1,TABSIZ    .
            L,U         A5,2         . CONSTANT TO FILL TABLE
            L           X6,(0,A5)    . STARTING SOURCE REGISTER
            L           X1,(1,TABLE) . STARTING DEST. REGISTER
            BT          X1,0,*X6     . COPY 'COUNT' WORDS
```

The modifier part of X6 contains the address of A5, so the data (a whole word with the value 000000000002) will be transferred from there to TABLE. The increment portion of X6 is set up with a zero, so the modifier will not change as the instruction proceeds. (Note that you can transfer into or out of GRS locations, as in this example, but between GRS locations any partial-word modifier is still ignored; a whole word is transferred.)

This example transfers H2 of 12 words of TAB1 to TAB2. H1 of each word of TAB2 remains unchanged:

```
TABSIZ      $EQU        100          . NUMBER OF ENTRIES TO RESERVE
TABENT      $EQU        2            . ENTRY SIZE
COUNT       +           12           . REPEAT COUNT
TAB1        $RES        TABSIZ*TABENT . TABLE 1
TAB2        $RES        TABSIZ*TABENT . TABLE 2
            L           R1,COUNT     . GET REPEAT COUNT
            L           X5,(1,TAB1)  . STARTING SOURCE REGISTER
            L           X1,(1,TAB2)  . STARTING DEST. REGISTER
            BT,H2       X1,0,*X5     . COPY 'COUNT' WORDS
```

15.2 Search Instructions

A number of other instructions, like Block Transfer, use the repeat count register, R1, to control the same operation on multiple words. These search and masked search instructions test the contents of A_a against the contents of U. If the test is satisfied, the instruction skips NI (the next instruction). If the test is not satisfied, the test is repeated until the repeat count is exhausted, in which case NI is executed.

Here are formal definitions of the search instructions:

- Search Equal (SE) - Skips NI if the contents of U equals the contents of A_a, else repeats.
- Search Not Equal (SNE) - Skips NI if the contents of U are not equal to the contents of A_a, else repeats.
- Search Less Than or Equal (SLE)/ Search Not Greater (SNG) - Skips NI if the contents of U are less than or equal to the contents of A_a, else repeats.

- Search Greater (SG) - Skips NI if the contents of U are greater than the contents of A_a, else repeats.

What is accomplished by repeating the test? Well, normally, you will specify an X register and index incrementation (h bit set) on the instruction. In this manner, you can test successive entries of a table. (Of course, you must know how many entries to test!)

The search instructions allow you to specify a j partial-word modifier, testing only part of each word. If you use a partial-word modifier, remember that the value in A_a is tested against the *right-justified* partial-word value. In other words, if you want to search for a word with 017 in S3, put 017 in A5 and code "SE,S3 A5,0,*X9" (as an example).

In practice, the instruction following a search or masked search is usually a jump. This allows passing control away from the search and the code which processes the location found by the search.

If you have trouble remembering which algebraic search does which test (less than, greater than, etc.), use this mnemonic device: Just like the test instructions (Test Less Than or Equal, Test Greater), the search instructions test (U) *relation* (A); U always comes first in the relation.

Here is an example of how to use a search instruction:

```
ENTSIZ    $EQU      3                 . ENTRY SIZE 3 WORDS
MAXENT    $EQU      20                . MAX ENTRY COUNT
ENTCNT    +         0                 . ACTUAL ENTRY COUNT
NAMETAB   $RES      ENTSIZ*MAXENT     . NAME TABLE
NTNAME    $EQUF     NAMETAB,,W        . NAME FIELD (2 WORDS)
NTOTHER   $EQUF     NAMETAB+2,,W      . OTHER INFORMATION
.
          L         A6,GOODNM         . NAME TO SEARCH FOR
          L         A1,ENTCNT         . GET ACTUAL ENTRY COUNT
          L         X5,(+ ENTSIZ,0)   . STARTING INDEX VALUE
          SE        A6,NTNAME,*X5     . LOOK FOR NAME
          J         ALG090            . NO-FIND,ALG090 FOUND,
          AN,U      X5,ENTSIZ         . BACK UP ONE ENTRY
          L         A1,NTOTHER,X5     .USE OTHER INFO IN ENTRY
```

(The actual entry count must be filled in at execution time before entering this code.) Note the line labelled "back up one entry". This operation is necessary because of the way U, the operand address is generated. Little u, from the instruction, is added to the modifier portion of Xx to create U. The contents of U are tested against A_a; only then is the index incrementation done, adding the increment portion of X_x to the modifier portion to create a new modifier. The CPU does this index incrementation regardless of *whether the comparison is successful.* As a result, when the match is found, the index register is incremented, causing it to point one entry past the matching entry. By means of a little simple arithmetic, we can avoid backing up, changing a few lines in the above example:

```
          L         X5,(+ ENTSIZ,-ENTSIZ) . STARTING INDEX
          SE        A6,NTNAME+ENTSIZ,*X5  . LOOK FOR NAME
          J         ALG090            . NO-FIND,ALG090 FOUND,
          L         A1,NTOTHER,X5     .
```

This change simply adjusts the balance between starting value of the index register and the u address on the SE instruction. The search will begin looking the first entry, just as before. (NTNAME+ENTSIZ-ENTSIZ equals NTNAME.) And just as before, it will step through the entries, one at a time. Only now, when the incrementation after the match takes place, X5 will be left pointing directly to the matching entry. This technique takes advantage of index register incrementation, instead of fighting it.

15.3 More Search Instructions

Another pair of instructions tests the contents of U against pairs of A registers. Here are their descriptions:

- Search Within Range (SW) - skips NI if the contents of U are greater than A_a and less than or equal to A_{a+1}, else repeats.
- Search Not Within Range (SNW) - skips NI if the contents of U are less than or equal to A_a or greater than A_{a+1}.

The formal description of SW says that it skips NI if "$(A_a)<(U)\leq(A_{a+1})$". Note that the test is for the contents of A_a strictly less than the contents of U. But the comparison between the contents of U and A_{a+1} is less than or *equal to*. This is identical to the test done by Test Within Range (TW). See Chapter 8 for more on how to set up comparison values for "within range" tests.

These instructions use the index register in the same way as Search Equal, Search Not Equal, and the others described above. The same suggestions for index setup apply for SW and SNW, as well.

15.3.1 Restarting A Search Loop

More often than not, a search step may need to locate all the matching entries in a table, not just the first. In that case, restarting is not hard. The CPU decrements the repeat count register (R1) each time it performs a test. When a match is found, the value left in R1 accurately reflects the number of remaining entries. As long as you don't change the index register used by the search, you should be able to jump right back to the search instruction itself to pick up where it was.

This listing of the example used before shows the restart:

```
ENTSIZ    $EQU      3                . ENTRY SIZE 3 WORDS
MAXENT    $EQU      20               . MAX ENTRY COUNT
ENTCNT    +         0                . ACTUAL ENTRY COUNT
NAMETAB   $RES      ENTSIZ*MAXENT    . NAME TABLE
NTNAME    $EQUF     NAMETAB,,W       . NAME FIELD
NTOTHER   $EQUF     NAMETAB+2,,W     . OTHER INFORMATION
.
          L         A6,GOODNM        . NAME TO SEARCH FOR
          L         A1,ENTCNT        . GET ACTUAL ENTRY COUNT
          L         X5,(+ ENTSIZ,-ENTSIZ) . STARTING INDEX
ALG070    .
          SE        A6,NTNAME+ENTSIZ,*X5 . LOOK FOR NAME
          J         ALG090           . NO-FIND,ALG090 FOUND,
          L         A1,NTOTHER,X5    .
          .
          . (USE THE ENTRY JUST FOUND.)
          .
          J         ALG070           .
```

Remember that the code to use the match entry must not change R1, either.

(A debugging note: a repeat count which is too high, say, starting with alphabetic data—a bug—easily can result in a guard mode contingency. If you get an IGDM and the reentry address points to a search instruction, check where you are getting the repeat count from. Likewise, a faulty starting index can result in a false "no find" condition.)

Since there is no double-word search instruction, if you must search for a two-word key, you must make appropriate changes in the code above. After the no-find jump to ALG090, insert:

```
          TE        A7,NTNAME+1,X5   . SECOND WORD MATCH?
          J         ALG070           .  NO,LG070  YES,
```

This code assumes that you have set up the second word of the search name in A7.

15.4 Masked Search Instructions

While the search instructions just described can test any j-defined partial word, that is all they can do. The masked search instructions, in contrast, test an entire word, but only bits corresponding to those set in R2, the mask register. (Once again, R2 is hardwired for that function.) Here are the formal descriptions:

- Masked Search Equal (MSE) - skips NI if the contents of U ANDed with the contents of R2 equal the contents of A_a ANDed with the contents of R2, else repeats.
- Masked Search Not Equal (MSNE) - skips NI if the contents of U ANDed with the contents of R2 are not equal to the contents of A_a ANDed with the contents of R2, else repeats.
- Masked Search Less Than or Equal (MSLE)/ Masked Search Not Greater (MSNG) - skips

NI if the contents of U ANDed with the contents of R2 are less than or equal to the contents of A_a ANDed with the contents of R2, else repeats.

- Masked Search Greater (MSG) - skips NI if the contents of U ANDed with the contents of R2 are less than or equal to the contents of A_a ANDed with the contents of R2, else repeats.

These instructions are admittedly quite a mouthful. In practice, the author has had little use for them. Once you understand how to use the other search instructions, and understand the principle of the mask register, R2, generating a masked search is not difficult.

This example should be sufficient. It shows how to search a table for entries which have bits 31 (4) and 8 (27) set in the first word of the entry:

```
ENTSIZ    $EQU      2                 . ENTRY SIZE 3 WORDS
MAXENT    $EQU      20                . MAX ENTRY COUNT
ENTCNT    +         0                 . ACTUAL ENTRY COUNT
MASKTAB   $RES      ENTSIZ*MAXENT     . MASK TABLE
MTMASK    $EQUF     MASKTAB,,W        . NAME FIELD
MTOTHER   $EQUF     MASKTAB+1,,W      . OTHER INFORMATION
SRMASK    +         1*/2+1*/4+1*/27+1*/32 . COMP MASK
TSMASK    +         1*/4+1*/27        . TEST MASK
.
          L         A6,SRMASK         . COMPARISON MASK
          L         A1,ENTCNT         . GET ACTUAL ENTRY COUNT
          L         R2,TSMASK         . GET SEARCH MASK
          L         X5,(+ ENTSIZ,-ENTSIZ) . STARTING INDEX
MSK070    .
          MSE       A6,MTMASK+ENTSIZ,*X5 . LOOK FOR MASK
          J         MSK090            . NO-FIND,MSK090 FOUND,
          L         A1,MTOTHER,X5     .
          .
          . (USE THE ENTRY JUST FOUND.)
          .
          J         MSK070            .
```

Note that there are bits set in A6, the test register, besides those of concern (31 (4) and 8(27)). Since only bits 31 (4) and 8 (27) are set in the mask register, the other bits in A6 are ignored.

Exercises

E15-1. Code the $EQUs and other statements to copy the first 25 words of LIST10 to LIST99. Make sure that the 25 is a $EQU, so you can change it later.

E15-2. The intervals between source and destination words need not be the same. Assume there are two tables. UNITLS has 4-word entries, and PACKS has 7-word entries. There are UCOUNT entries in each list. Code a block transfer to copy H1 of the first word of a UNITLS entry to H1 of the first word of a PACKS entry. (Hint: choose increment portion of source and destination registers carefully.)

E15-3. Using your definition of UNITLS from the previous exercise, write code to find the entry which has words 1 and 2 equal to Fieldata 'LONGFILENAME'. (Hint: use SE to locate the first word; test the second word separately.)

Chapter 16: MASM Procedures

16.1 Internal Procedures - $PROC

A procedure is a set of MASM directives and assembly lines which can be referenced ("called") once or many times. Each time it is referenced, it delivers as its output (within the assembly) instructions in memory, data in memory or dictionary entries (label definitions), or all of these. Procedures can be nested and can make decisions based on parameters in the procedure reference line. (By the way, procedures are also known informally as "procs", after the MASM directive which defines a procedure. This text will use the terms interchangeably.)

The minimum directives needed to define a procedure are:

```
<plabel>*  $PROC
           .
           . (body of the proc)
           .
           $END
```

where:

- <plabel> is the name of the proc. The proc name is subject to the same limitations as any other MASM label (containing only A-Z, 0-9, _, and $, and beginning with an alpha character). The * (asterisk) is necessary to define the name of the proc outside the subassembly which defines the proc. In other words, if <plabel> is to be a proc, you must have the asterisk.

- $END is the end of the proc definition. If you omit the $END directive, MASM will consider the rest of the source element to be part of the proc, probably not what you had in mind.

The body of the proc may consist of zero or more: directives, instructions, data-generating lines or references to other procs. Normally, any instructions or data generated by the proc will go into the same location counter in effect where the proc was referenced. A later section of this chapter will explore location counter changes.

References to a proc have the form:

```
<label>    <plabel>   <params>
```

where:

- <label> is the statement label. A label on an instruction or data-generating line is assigned the current value of the current location counter. But this action does not take place on a proc reference line. You can control the value assigned to <label> by using a *waiting label*, defined later in this chapter, section 16.3.

- <plabel> is the proc name, like GENERATE or TABENT.

- <params> are the proc parameters. You can also put parameters in the operation field after the proc label, separated by commas (that is, something like TABENT,5,'Y').

What are procs good for? They allow you to capsulize a sequence of instruction or data lines, generating that sequence as often as necessary. For example, the calling sequence for ER PRINT$ is:

```
        L       A0,<packet>
        ER      PRINT$
```

If your program has to call PRINT$ a lot, you could get tired of coding the same pair of instructions over and over. One way to save time is to code a simple proc to generate the calling sequence:

```
ERPRT$* $PROC
        L       A0,ERPRT$(1,1)  .
        ER      PRINT$          .
        $END                    .
```

You can use ERPRT$ like this:

```
        ERPRT$  PKT17.
```

The construct ERPRT$(1,1) (in the second line of the proc) is a procedure reference form, or paraform. It obtains the value of the parameter on the proc reference line. In the example, that parameter is PKT17. A later section will go into more detail about proc parameters.

Does this proc seem suspiciously like P$RINT? Yes, it does work exactly the same as P$RINT. As it happens, the definition of P$RINT is part of a proc which defines a number of ER calls at once. You can see it for yourself in SYS$*RLIB$.PROC$/SYS75R2B. (The version name is dependent on the exact release of SYSLIB used by your site.) Procs and other code in the system library are not magic, and you should not be afraid to look at them. You can learn tricks you can use yourself.

The next proc is not as trivial. It generates the text and packet for a canned message to be printed with APRINT$. It assumes the form PF is defined elsewhere as 12, 6, and 18 bits:

```
ERRMSG* $PROC
Q       ERRMSG(1,1)             .
*       PF      2,$-Q,Q         .
        $END                    .
```

This is a sample use of ERRMSG:

```
BADPRM  ERRMSG  'PARAMETER 1 IS NOT NUMERIC'     .
```

Reference the resulting packet the same way as any other APRINT$ packet:

```
        A$PRINT BADPRM      .
```

Once you have hand-coded more than a few APRINT$s, you will appreciate the time you can save with a proc like this. The only flexibility you lose using this proc is that the number-of-lines

parameter is fixed at 2 (that is, double spacing after the previous print line). Nearly every program coded by the author uses this proc, or one like it.

16.2 The $NAME Directive

The $NAME directive creates an additional entry point into a procedure. Using $NAMEs, you can create one proc with multiple entry points. Or, a proc may have only one name, defined by a $NAME directive. The parameter references still refer to the proc name. Procs defined using $NAME directives for additional entry points typically don't use the name on the $PROC directive as an entry point; they omit the * on the label field. This is the author's usual definition of the ERRMSG proc:

```
P           $PROC
ERRMSG*     $NAME
Q           P(1,1)
*           PF          2,$-Q,Q
            $END
```

The big advantage of using $NAMEs as proc entry points is the dummy entry point on the proc, P in this case. You can make the $NAME entry point as long and descriptive as necessary, and the parameter references short, like P(1,1). A single reference, as in this example, is not taxing. But when there are many references in a single line of the proc, a shorter name is easy to appreciate.

As long as you never have a "P* $PROC" directive at the beginning of a proc, you can use "P $PROC" (no asterisk) to begin every proc you use. There will be no conflict.

Once again, the $NAME entry point must have an asterisk to define the label outside the proc.

System-defined procs use a trick you can try someday. They use the value appearing on the $NAME directive to "remember" the label by which the proc was entered, something like this:

```
P           $PROC
P$RINT*     $NAME       PRINT$
A$PRINT*    $NAME       APRINT$
            L           AO,P(1,1)
            ER          P(0,0)
            $END
```

The parameter reference P(0,0) supplies the name of the ER associated with the entry point by which the proc was called. If the proc is called by P$RINT, the ER called will be PRINT$; if by A$PRINT, the ER will be APRINT$.

16.2.1 Parameter References

Figures 16-1a and 16-1b on the following page show a sample proc and parameter references based on it. Here are the parameter references:

P is the number of fields on the proc reference plus 1, in this case, 3. (If the proc were not entered via a NAME$ label, P would be 2.) This value includes the operation field (in this case, "TABENT,5,8").

P(0,0) is the value on the $NAME directive by which the proc was entered, in this case 15.

P(0,1) is the first parameter in the operation field, in this case 5. P(0,2) is the second parameter in the operation field, 8.

P(1) is the number of subfields in the first field, 2 in this case. P(2) is the number of subfields in the second field, 3.

P(1,1) is the value in the first field, first subfield, in this case LAB1. P(1,2) is the value in the second subfield of the first field, 27.

P(2,1) is the value in the second field, first subfield, in this case TAG15. P(2,2) is the value in the second field, second subfield, 12.

```
P          $PROC
TABENT*    $NAME  15
           +      P,P(0,0)
           +      P(1),P(1,1),P(1,2)
           +      P(2,1),P(2,2),P(2,3)
           $END
```

Figure 16-1a. Proc Definition

There is no practical limit to the number of fields you can put on a proc reference. A field on the $PROC directive allows you to impose a limit yourself, if necessary.

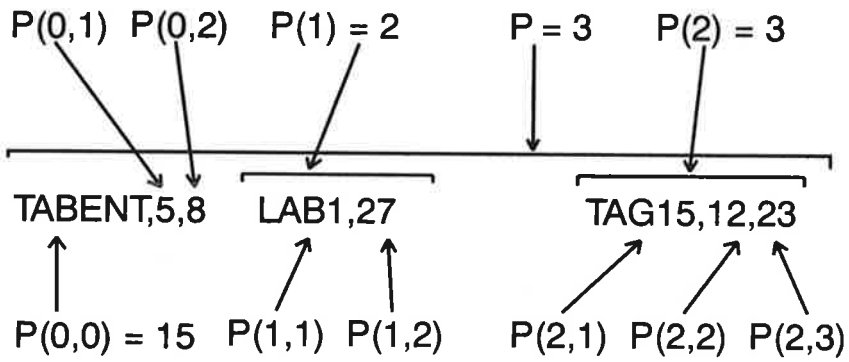

Figure 16-1b. Proc Reference.

If a proc references a subfield not defined by a particular proc reference, MASM will supply a value of zero. In the case of the proc in Figure 16-1b, a reference to P(1,3) would get a value of zero. If a particular subfield is important to your proc, your proc must check for it explicitly; MASM will not do it for you.

16.3 External and Waiting Labels

Labels inside a proc are handled a little differently from labels outside a proc. The code inside a proc is a "subassembly"; MASM begins a new, lower level of its dictionary, or list of label definitions. Labels defined at a higher level than the proc (that is, the main assembly) are visible to the proc, but not the reverse. Use an asterisk on a label to define it at the next level up. Each additional asterisk defines the label at a higher level still. The limit of externalization is to define a label as an entry point in the relocatable element produced by MASM. Take this proc as an example:

```
P           $PROC.
ENT1*       $NAME.
            .
TAG42**     .
            .
            $END
```

When this proc is referenced, it will define TAG42 as an entry point in the relocatable output element produced by MASM, just as if you had coded TAG42* as a label in the main assembly. You must take care not to reference this proc but once in an element, or D (duplicate) error flags will result.

The ERRMSG proc defined earlier uses a device called a waiting label to define the label on a proc reference line. Here is the proc definition again, for your convenience:

```
P           $PROC
ERRMSG*     $NAME
Q           P(1,1)
*           PF          2,$-Q,Q
            $END
```

The asterisk on the fourth line of the proc definition causes MASM to define the label on the proc reference line as the current value of the current location counter. Given this reference to the proc:

```
BADPRM      ERRMSG      'PARAMETER 1 IS NOT NUMERIC!'.
```

The label BADPRM is given the address of the APRINT$ packet (PF 2,$-Q,Q). (Note that since the label Q has no asterisk, it exists only for the duration of the proc subassembly. It will not conflict with any name in the main assembly or with itself.)

You might even want to make a waiting label an entry point in the output element. Inside the proc, asterisks in columns 1 and 2 will do the trick. The first asterisk makes the waiting label, the second externalizes it.

16.4 External Procedures

Defining a proc to be used by more than one assembly requires help from the Procedure Definition Processor, PDP. PDP processes proc elements for MASM, Cobol, and Fortran. PDP does not "know" anything about MASM beyond the $PROC, $NAME, and $END directives. Code lines between $PROC and $END pairs are made available to MASM without editing. PDP accepts a source element containing only proc definitions, locates the procs, and stores the element with pointers that help MASM find the procs.

What follows is a sample runstream to define a proc using PDP:

```
@PDP,IL     MYFILE.MYPROCS
P           $PROC                    .
ERRMSG*     $NAME                    .
Q           P(1,1)                   .
*           PF          2,$-Q,Q      .
            $END                     .
.
P           $PROC                    .
C$SF*       $NAME       CSF$         .
A$CSF*      $NAME       ACSF$        .
            L,U         A0,P(1,1)    .
            ER          P(0,0)       .
            $END
@EOF
```

(Here is an example use of this proc:

```
            C$SF        ('@ASG,T TDIR$.,F///1000 . ') .
```

The parentheses around the string create a literal. The space period space terminator at the end of the image is suggested, because this proc gives the ER CSF$ a zero length, in which case the Exec assumes the image is 14 words; 20 words for ACSF$.)

The I option, as for any other system processor, tells PDP to accept source code from the runstream and insert into the element named in spec 1 (MYFILE.MYPROCS, in this case). The L option tells PDP to list the source lines as they are stored.

If you prefer, you can create the source element using CTS or EDIT-1100, then get PDP to process it as a proc element:

```
@PDP,N      MYFILE.MYPROCS,.MYPROCS
@EOF
```

Absence of the I option tells PDP to read the element in spec 1, process it as a proc, and put the output into the element named in spec 2. The element names in spec 1 and 2 may be different, or as in this case, the same. Since the proc element type is the same as a symbolic element, specifying the same element name twice causes the input element to be destroyed. You can edit the new element, but doing so makes it no longer a proc element; you'll have to put it back through PDP again.

The N option tells PDP not to list the source.

16.5 Proc File Search Order

When you code a proc reference in your MASM code, and the proc is not defined within the source element, MASM has to look elsewhere. MASM has several places it can search for proc definitions:

1. The current assembly,
2. MASM$PF, if it is assigned to the run (usually an @USE name),
3. ASM$PF, if it is assigned to the run (usually an @USE name),